Construction and Verification of Software

2019 - 2020

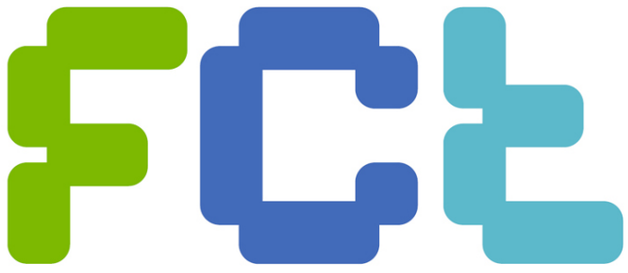
MIEI - Integrated Master in Computer Science and Informatics

Consolidation block

Lecture 4 - Abstract Data Types

João Costa Seco (joao.seco@fct.unl.pt)

based on previous editions by **Luís Caires** (lcaires@fct.unl.pt)



FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Outline

- Loop Invariants (recap) Sorted Arrays
- Sorting algorithms
- Abstract Data Types
- Verifying Classes and Objects
- Abstract and Representation State
- Soundness of State Mapping

Construction and Verification of Software

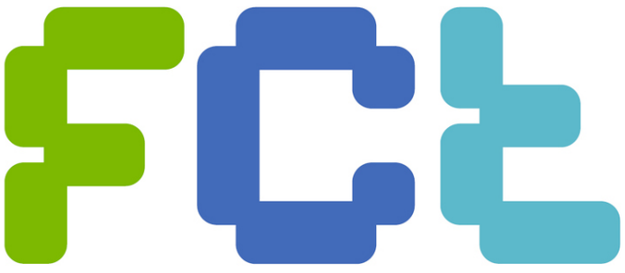
2019 - 2020

MIEI - Integrated Master in Computer Science and Informatics

Consolidation block

Lecture 4 - Part I - Loop Invariants and Sorting

João Costa Seco (joao.seco@fct.unl.pt)
based on previous editions by **Luís Caires** (lcaires@fct.unl.pt)



FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Loop Invariants

- Loop invariant approximate state assertions before the loop, between iterations, and at the end of the loop.

```
function maxArray(a:array<int>, n:int, m:int):bool
    requires 0 < n <= a.Length
    reads a
{ forall k : int :: 0 <= k < n ==> a[k] <= m }

method Max(a:array<int>) returns (m:int)
    requires 0 < a.Length
    ensures maxArray(a,a.Length,m)
{
    m := a[0];
    var i := 1;
    while i < a.Length
        invariant 1 <= i <= a.Length
        invariant maxArray(a,i,m)
    {
        if m < a[i]
        { m := a[i]; }
        i := i + 1;
    }
}
```

Example: BinarySearch

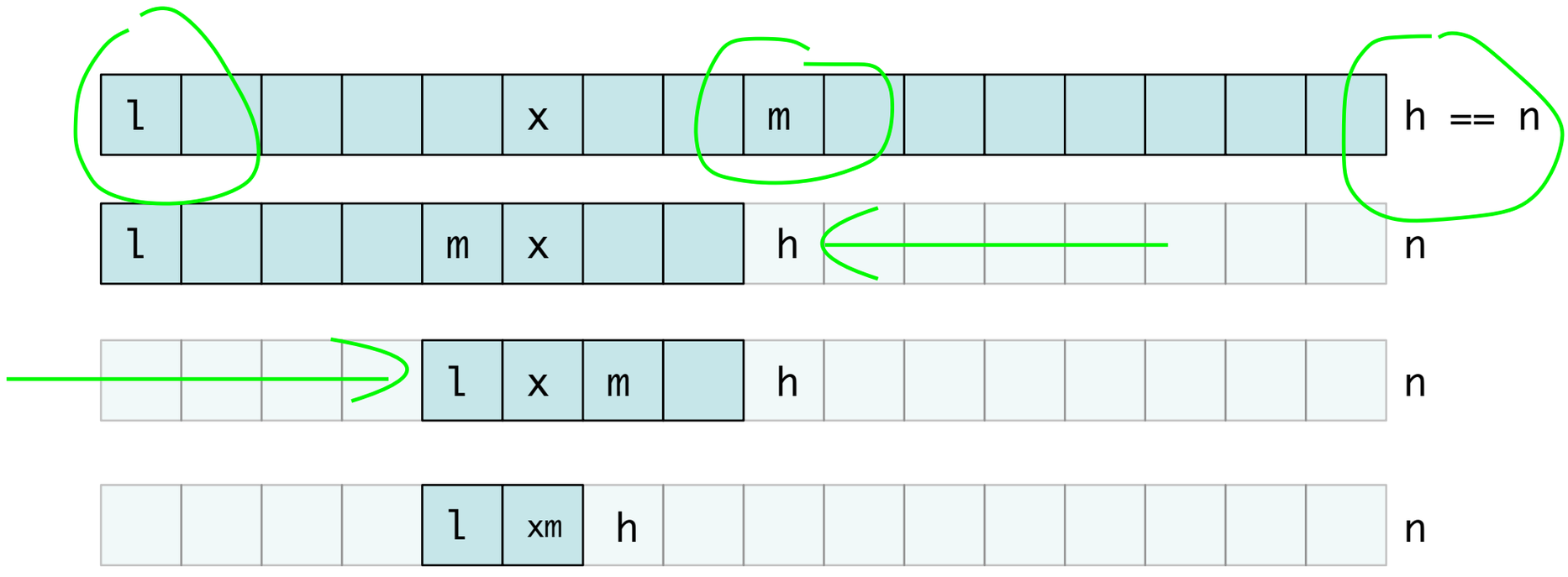
```
function sorted(a:array<char>, n:int):bool
    requires 0 <= n <= a.Length
    reads a
{   forall i, j:: (0 <= i < j < n) ==> a[i] <= a[j]  }
```

Example: BinarySearch

```
function sorted(a:array<char>, n:int):bool
    requires 0 <= n <= a.Length
    reads a
{   forall i, j:: (0 <= i < j < n) ==> a[i] <= a[j]  }

method BSearch(a:array<char>, n:int, value:char) returns (pos:int)
    requires 0 <= n <= a.Length && sorted(a, n)
    ensures …
    ensures …
{

}
```

Example: BinarySearch

```
function sorted(a:array<char>, n:int):bool
    requires 0 <= n <= a.Length
    reads a
{   forall i, j:: (0 <= i < j < n) ==> a[i] <= a[j]  }

method BSearch(a:array<char>, n:int, value:char) returns (pos:int)
    requires 0 <= n <= a.Length && sorted(a, n)
    ensures 0 <= pos ==> pos < n && a[pos] == value
    ensures pos < 0  ==> forall i :: (0<= i < n) ==> a[i] != value
{
```




```
}
```

Example: BinarySearch

```
function sorted(a:array<char>, n:int):bool
    requires 0 <= n <= a.Length
    reads a
{ forall i, j:: (0 <= i < j < n) ==> a[i] <= a[j] }

method BSearch(a:array<char>, n:int, value:char) returns (pos:int)
    requires 0 <= n <= a.Length && sorted(a, n)
    ensures 0 <= pos ==> pos < n && a[pos] == value
    ensures pos < 0  ==> forall i :: (0<= i < n) ==> a[i] != value
{
    var low, high := 0, n;
    while low < high
        decreases high - low
        invariant ???
        invariant ???
        invariant ???
    {
        var mid := (low + high) / 2;
        if a[mid] < value           { low := mid + 1; }
        else if value < a[mid]      { high := mid; }
        else /* value == a[mid] */ { return mid; }
    }
    return -1;
}
```

Example: BinarySearch

```
function sorted(a:array<char>, n:int):bool
    requires 0 <= n <= a.Length
    reads a
{ forall i, j:: (0 <= i < j < n) ==> a[i] <= a[j] }

method BSearch(a:array<char>, n:int, value:char) returns (pos:int)
    requires 0 <= n <= a.Length && sorted(a, n)
    ensures 0 <= pos ==> pos < n && a[pos] == value
    ensures pos < 0  ==> forall i :: (0<= i < n) ==> a[i] != value
{
    var low, high := 0, n;
    while low < high
        decreases high - low
        invariant 0 <= low <= high <= n
        invariant forall i :: 0 <= i < n && i < low ==> a[i] != value
        invariant forall i :: 0 <= i < n && high <= i ==> a[i] != value
    {
        var mid := (low + high) / 2;
        if a[mid] < value           { low := mid + 1; }
        else if value < a[mid]      { high := mid; }
        else /* value == a[mid] */ { return mid; }
    }
    return -1;
}
```

Construction and Verification of Software

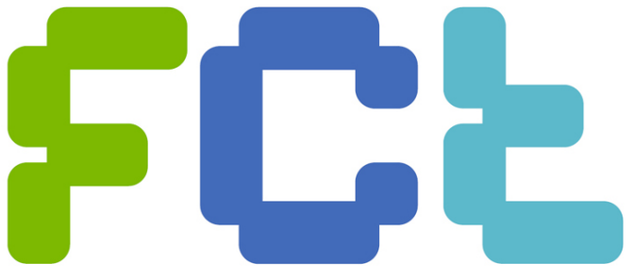
2019 - 2020

MIEI - Integrated Master in Computer Science and Informatics

Consolidation block

Lecture 4 - Part II - Sorting

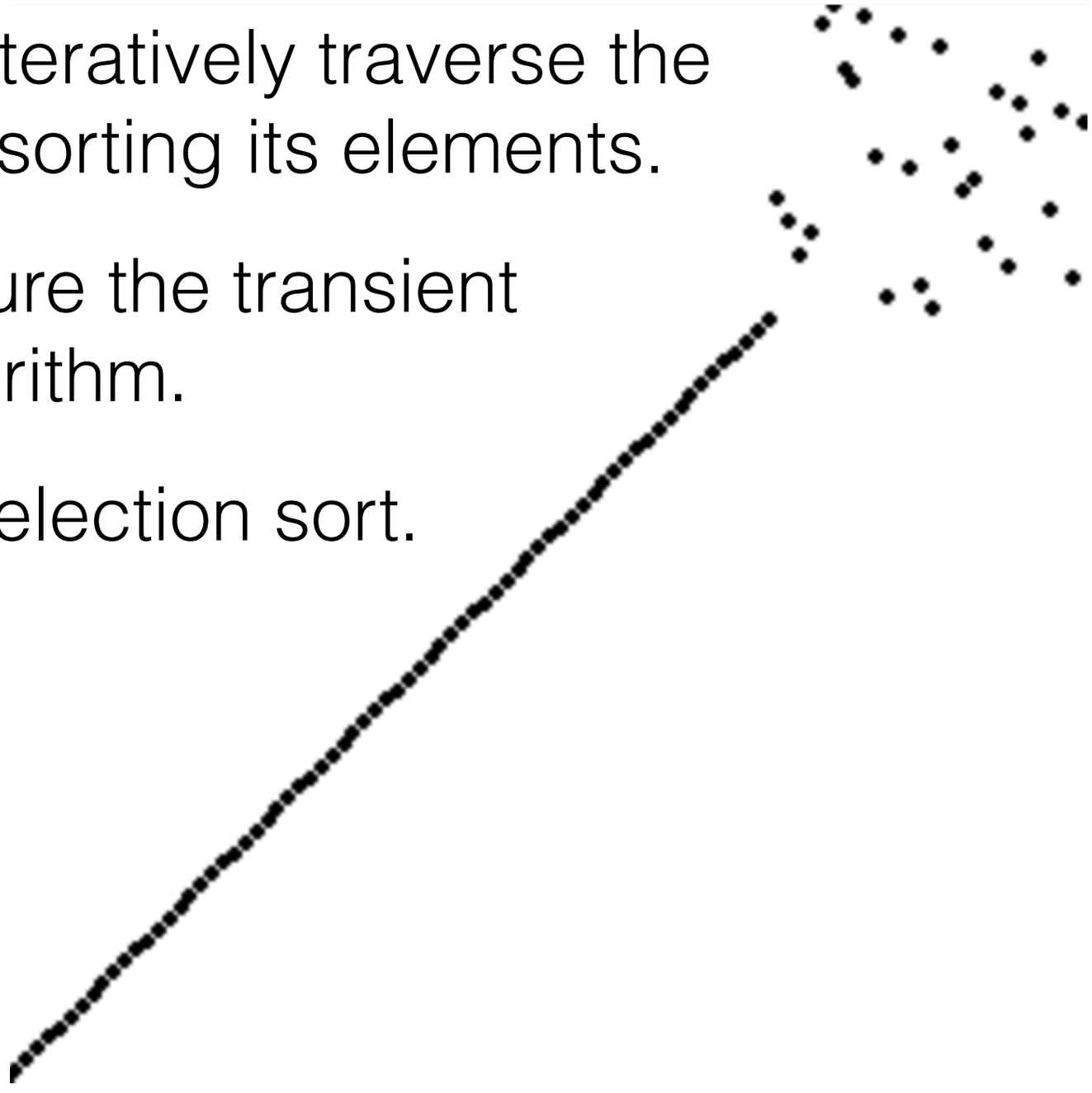
João Costa Seco (joao.seco@fct.unl.pt)
based on previous editions by **Luís Caires** (lcaires@fct.unl.pt)



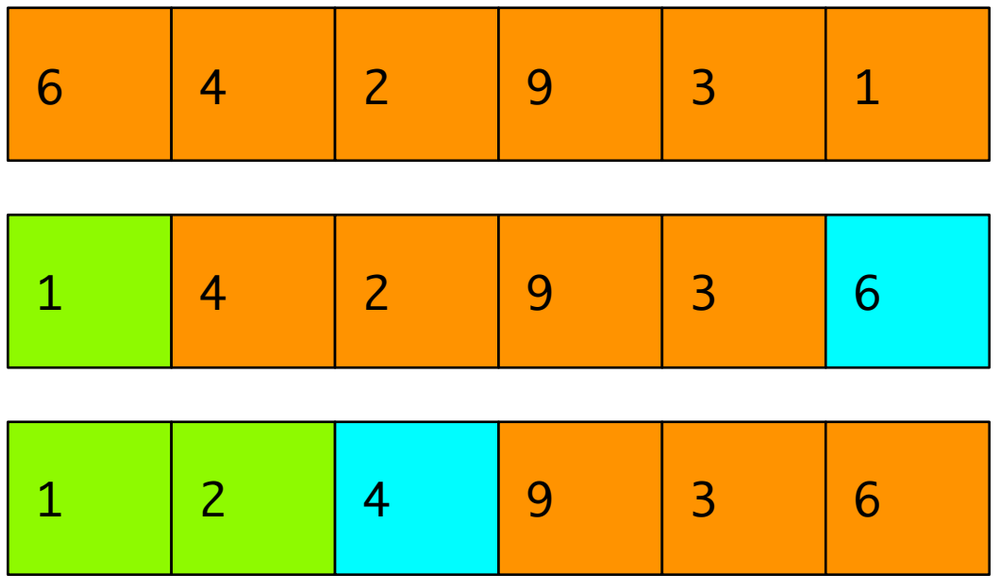
FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

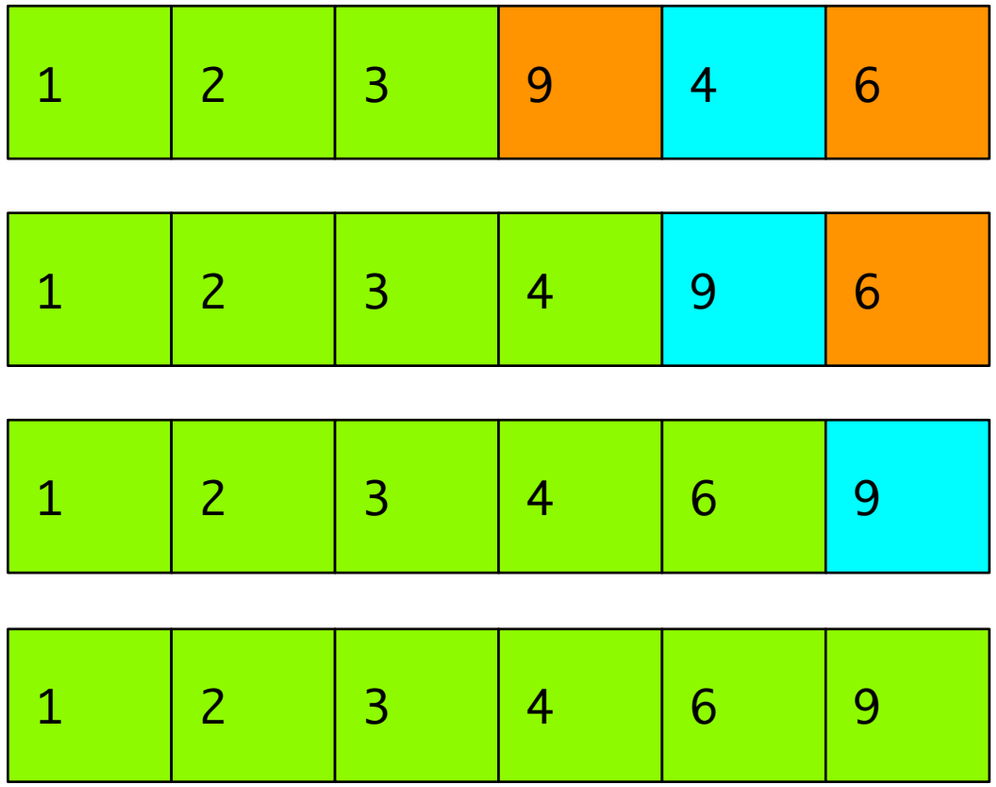
Sorting — Rationale

- Most sorting algorithms iteratively traverse the data structure gradually sorting its elements.
- The loop invariants capture the transient status of the sorting algorithm.
- We illustrate that using selection sort.



Selection Sort

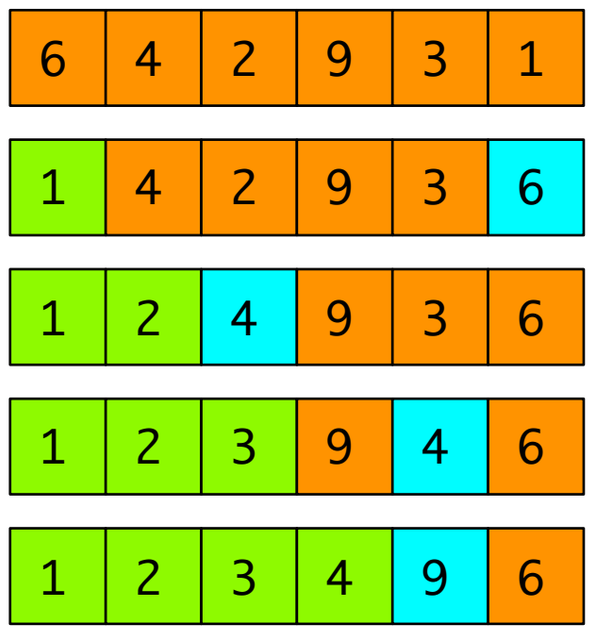




Selection Sort

- Some aux functions

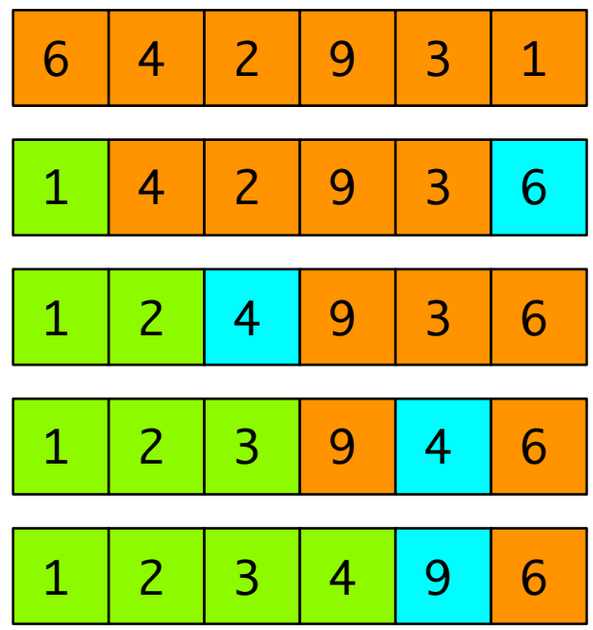
```
function sorted(a:array<char>, n:int):bool
    requires 0 <= n <= a.Length
    reads a
{ forall i, j:: (0 <= i < j < n) ==> a[i] <= a[j] }
```




```
function partitioned(a:array<char>,i:int,n:int):bool
    requires 0 <= n <= a.Length
    reads a;
{ forall k, l :: 0 <= k < i <= l < n ==> (a[k] <= a[l]) }
```

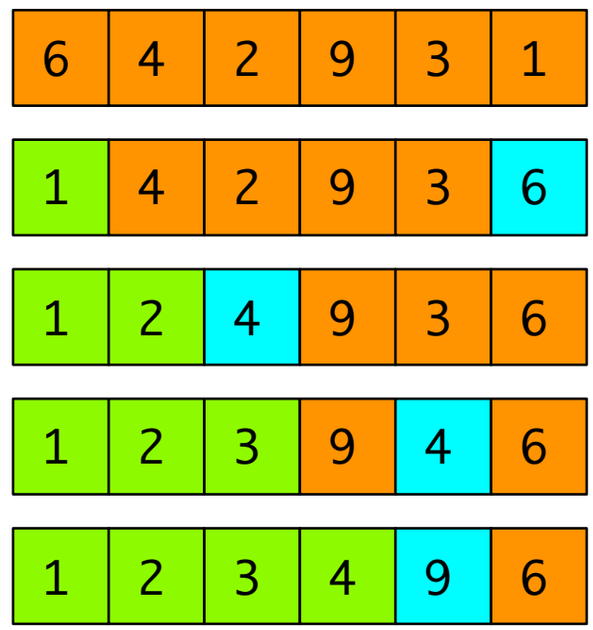
Selection Sort

```
method selectionSort(a:array<char>, n:int)
{
  var i := 0;
  while i < n {
    selectSmaller(a, i, n);
    i := i + 1;
  }
}
method selectSmaller(a:array<char>, i:int, n:int)
{
  var j := i+1;
  while j < n  {
    if a[j] < a[i]
    {
      a[i], a[j] := a[j], a[i];
    }
    j := j + 1;
  }
}
```



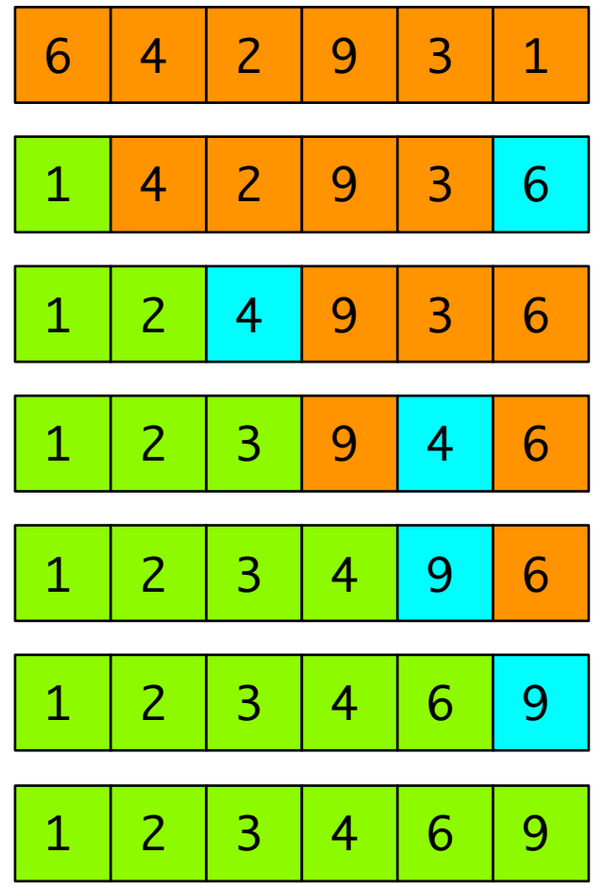
Selection Sort

```
method selectionSort(a:array<char>, n:int)
{
  var i := 0;
  while i < n
    decreases n - i
    invariant 0 <= i <= n
    invariant sorted(a, i)
    invariant partitioned(a, i, n)
  {
    selectSmaller(a, i, n);
    i := i + 1;
  }
}
method selectSmaller(a:array<char>, i:int, n:int)
  requires 0 <= i < n <= a.Length
  requires sorted(a, i)
  requires partitioned(a, i, n)
  modifies a
  ensures sorted(a, i+1)
  ensures partitioned(a, i+1, n)
```



Selection Sort

```
method selectSmaller(a:array<char>, i:int, n:int)
  requires 0 <= i < n <= a.Length
  requires sorted(a, i)
  requires partitioned(a, i, n)
  modifies a
  ensures sorted(a, i+1)
  ensures partitioned(a, i+1, n)
{
  var j := i+1;
  while j < n
    decreases n - j
    invariant i < j <= n
    invariant a[..i] == old(a[..i])
    invariant forall k :: i < k < j ==> a[i] <= a[k]
    invariant sorted(a, i)
    invariant partitioned(a, i, n)
  {
    if a[j] < a[i]
    {
      a[i], a[j] := a[j], a[i];
   }
    j := j + 1;
  }
}
```



Exercise

- orderedInsert

```
method insert(a:array<char>, i:int, elem:char, n:int)
  requires 0 <= n < a.Length
  requires sorted(a, n)
  modifies a
  ensures sorted(a, n+1)
```

Construction and Verification of Software

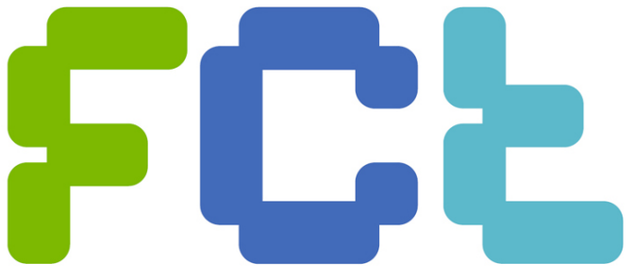
2019 - 2020

MIEI - Integrated Master in Computer Science and Informatics

Consolidation block

Lecture 4 - Part III - Abstract Data Types (Intro)

João Costa Seco (joao.seco@fct.unl.pt)
based on previous editions by **Luís Caires** (lcaires@fct.unl.pt)



FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Abstract Data Types (Liskov, 78)

- ADTs are the building blocks for software construction
  - Consists of:
    - A description of the data elements of the type
    - A set of operations over the data elements of the ADT
  - A software system is a composition of ADTs
  - ADTs behave like regular types in a programming language
  - Promotes modularity, encapsulation, information hiding, and hence reuse, modifiability, and correctness.

ADTs (Liskov & Zilles,78)

PROGRAMMING WITH ABSTRACT DATA TYPES

Barbara Liskov
Massachusetts Institute of Technology
Project MAC
Cambridge, Massachusetts

Stephen Zilles
Cambridge Systems Group
IBM Systems Development Division
Cambridge, Massachusetts

Abstract

The motivation behind the work in very-high-level languages is to ease the programming task by providing the programmer with a language containing primitives or abstractions suitable to his problem area. The programmer is then able to spend his effort in the right place; he concentrates on solving his problem, and the resulting program will be more reliable as a result. Clearly, this is a worthwhile goal.

Unfortunately, it is very difficult for a designer to select in advance all the abstractions which the users of his language might need. If a language is to be used at all, it is likely to be used to solve problems which its designer did not envision, and for which the abstractions embedded in the language are not sufficient.

This paper presents an approach which allows the set of built-in abstractions to be augmented when the need for a new data abstraction is discovered. This approach to the handling of abstraction is an outgrowth of work on designing a language for structured programming. Relevant aspects of this language are described, and examples of the use and definitions of abstractions are given.

Barbara Liskov (MIT)

BARBARA LISKOV
United States – **2008**

For contributions to practical and theoretical foundations of programming language and system design, especially related to data abstraction, fault tolerance, and distributed computing.

Abstract Data Type

Abstract types are intended to be very much like the built-in types provided by a programming language. The user of a built-in type, such as integer or integer array, is only concerned with creating objects of that type and then performing operations on them. He is not (usually) concerned with how the data objects are represented, and he views the operations on the objects as indivisible and atomic when in fact several machine instructions may be required to perform them. In addition, he is not (in general) permitted to decompose the objects. Consider, for example, the built-in type integer. A programmer wants to declare objects of type integer and to perform the usual arithmetic operations on them. He is usually not interested in an integer object as a bit string, and cannot make use of the format of the bits within a computer word. Also, he would like the language to protect him from foolish misuses of types (e.g., adding an integer to a character) either by treating such a thing as an error (strong typing), or by some sort of automatic type conversion.

Abstract Data Type (External View)

- External View
  - A public opaque data type (that clients will use)

    Note: opaque means = behaves as a primitive type
  - A set of operations on this data type
  - Operations must neither reveal, nor allow a client to invalidate the internal representation of the ADT
  - pre and post conditions on these operations must be expressed in terms of the abstract type (the only type known to the client)
  - This is why ADTs promote reuse, modifiability, and correctness: the developer can change the implementation anytime, without breaking contracts

Abstract Data Type (Internal View)

- Internal View
  - A **representation** data type (hidden from clients)
  - A set of operations on the representation data type
- ***important remarks***
  - A programmer must define the operations in such a way that the representation state (invisible to clients) is kept consistent with the intended abstract state
  - Pre-conditions on the public operations, expressed on the abstract state, must map into pre-conditions expressed in terms of the representation state
  - The same for post-conditions
  - At all times the concrete state must represent a well defined abstract state (otherwise something is wrong!)

Example (Positive Set ADT)

```
class PSet {
// an abstract set of positive numbers

    method new(sz:int) {…}
    // initializes the set ( e.g., Java constructor )

    method add(v:int) {…}
    // adds v to the set if space available

    function size() : int {…}
    // returns number of elems in the set

    function contains(v:int) : bool {…}
    // returns number of elems equal to v in the set

    function maxsize() : int {…}
    // returns max number of elems allowed in the set
}
```

Technical ingredients in ADT design

- The ***abstract state***
  - defines how client code sees the object
- The ***representation type***
  - chosen by the programmer to implement the ADT internals. The programmer is free to choose the implementation strategy (data-structures, algorithms). This is done at construction time.
- The ***concrete state***
  - in general, not all representation states are legal concrete states
  - a concrete state is a representation state that really represents some well-defined abstract state

Technical ingredients in ADT design

- The ***representation invariant***
  - the representation invariant is a condition that restricts the representation type to the set of (safe) concrete states
  - if the ADT representation falls outside the rep invariant, something is wrong (inconsistent representation state).
- The ***abstraction function***
  - maps every concrete state into some abstract state
- The ***operation pre-*** *and* ***post- conditions***
  - expressed for the representation type
  - also expressed for the abstract type (for client code)

Construction and Verification of Software

2019 - 2020

MIEI - Integrated Master in Computer Science and Informatics

Consolidation block

Lecture 4 - Part IV - Abstract Data Types (with objects)

João Costa Seco (joao.seco@fct.unl.pt)
based on previous editions by **Luís Caires** (lcaires@fct.unl.pt)

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Bank Account ADT

- Abstract State
  - the account balance (`bal`)
  - `bal` is of type `int` subject to the constraint (`bal >= 0`)

Bank Account ADT

- Representation type
  - an integer `bal`
  - in this simple case the representation type is the same as the abstract type
  - the true "meaning" of the representation and abstract types are different
  - not all operations on integers are valid on account balances (e.g., to multiply bank accounts)

Bank Account ADT

- Representation type
  - an integer `bal`
  - in this simple case the representation type is the same as the abstract type
  - the true "meaning" of the representation and abstract types are different
  - not all operations on integers are valid on account balances (e.g., to multiply bank accounts)
- Representation invariant
  - (`bal >= 0`)
  - this time, pretty simple

Example (Account)

```
class Account {
    var bal: int;

    function RepInv():bool
    // specifies the representation invariant
       reads this ;
    {
          bal >= 0
     }
    …
}
```

Example (Account)

```
class Account {
    var bal: int;

    function RepInv():bool
    // specifies the representation invariant
      reads this ;
    {
      bal >= 0
    }

    method Init()
        modifies this;
        ensures RepInv()
    { bal := 0; }
    …
}
```

Example (Account)

```
class Account {
    var bal: int;
…
    // All operations must require the representation invariant
    // All operations must ensure the representation invariant
    method deposit(v:int)
        modifies this;
        requires RepInv() && v >= 0
        ensures RepInv()
    { bal := bal + v; }

    method withdraw(v:int)
        modifies this;
        requires RepInv() && v >= 0
        ensures RepInv()
    { if (bal>=v) { bal := bal - v; } }
}
```

Example (Account)

```
class Account {
    var bal: int;
…
    function getBal():int
        reads this
    { bal }

    method withdraw(v:int)
        modifies this;
        requires RepInv() && 0 <= v <= getBal()
        ensures RepInv()
    { bal := bal - v;  }
}
```

Construction and Verification of Software

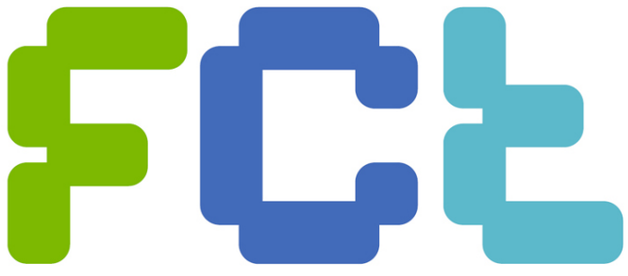
2019 - 2020

MIEI - Integrated Master in Computer Science and Informatics

Consolidation block

Lecture 4 - Part V - Abstract Data Types (an Example)

João Costa Seco (joao.seco@fct.unl.pt)
based on previous editions by **Luís Caires** (lcaires@fct.unl.pt)



FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Set ADT

```
class ASet {
// an abstract Set of numbers

    method new(sz:int) {}
    // initializes aset ( e.g., Java constructor )

    method add(v:int) {}
    // adds v to aset if space available )

    function size() : int
    // returns number of elems in aset

    function contains(v:int) : bool
    // check if v belongs to set

    function maxsize() : int
    // returns max number of elems allowed in aset

}
```

Set ADT

- Abstract State
  - a set of positive integers aset

Set ADT

- Representation type
  - an array of integers **store** with sufficient large size
  - an integer nelems counting the elements in **store**

Set ADT

- Representation type
  - an array of distinct integers **store**
  - an integer nelems counting the elements in **store**
- Representation invariant

```
(store != null) &&

(0 <= nelems <= store.length) &&

forall k :: (0<=k<nelements) ==> forall j::(k<j<nelements)  ==> b[k] != b[j]
```

Set ADT

- Representation type
  - an array of **distinct** integers **store**
  - an integer nelems counting the elements in **store**
- Representation invariant

```
(store != null) &&

(0 <= nelems <= store.length) &&

forall k :: (0<=k<nelements) ==> forall j::(k<j<nelements)  ==> b[k] != b[j]
```

Set ADT

- Representation type
  - an array of distinct integers **store**
  - an integer nelems counting the elements in **store**
- Representation invariant

```
(store != null) &&

(0 <= nelems <= store.length) &&

forall k :: (0<=k<nelements) ==> forall j::(k<j<nelements)  ==> b[k] != b[j]
```

- Abstraction mapping
  - <nelems=n, store=[v_0,v_1,…$v_{store.Length-1}$]> → {v_0,…,v_{n-1}}
  - more later ....

Set ADT

```
class ASet {

  var a:array<int>;
  var size:int;

  constructor(SIZE:int)
    requires SIZE > 0;
    ensures RepInv()
  {
    a := new int[SIZE];
    size := 0;
  }

…
```

Set ADT

```
class ASet {

  var a:array<int>;
  var size:int;

  constructor(SIZE:int)
    requires SIZE > 0;
    ensures RepInv()
  {
    a := new int[SIZE];
    size := 0;
  }

  function RepInv():bool
    reads this,a;
  {
   …
  }
…
```

Set ADT

```
class ASet {

  var a:array<int>;
  var size:int;

…

  function RepInv():bool
    reads this,a;
  {
    a!=null &&
    0 < a.Length &&
    0 <= size <= a.Length &&
    unique(a,0, size)
  }
…
```

Set ADT

```
class ASet {

  var a:array<int>;
  var size:int;

…
  function unique(b:array<int>, l:int, h:int):bool
  reads b;
  requires b != null && 0<=l <= h <= b.Length ;
  {
    forall k::(l<=k<h) ==> forall j::(k<j<h)  ==> b[k] != b[j]
  }
…
```

Set ADT

```
class ASet {

  var a:array<int>;
  var size:int;

  function count():int
  reads this,a;
  requires RepInv();
  { size }

  function maxsize():int
  reads this,a;
  requires RepInv();
  { a.Length }

  method add(x:int)
  modifies this,a;
  requires RepInv() && x >= 0 && count() < maxsize();
  ensures RepInv()
  {
    var f:int := find(x);
    if (f < 0) {
      a[size] := x;
      size := size + 1;
    }
  }
…
```

Set ADT

```
class ASet {

  var a:array<int>;
  var size:int;
…
  method find(x:int) returns (r:int)
  requires RepInv();
  ensures -1 <= r < size;
  ensures r < 0 ==> forall j::(0<=j<size) ==> x != a[j];
  ensures r >=0 ==> a[r] == x;
  {
    var i:int := 0;
    while (i<size)
    decreases size-i
    invariant 0<=i<=size;
    invariant forall j::(0<=j<i) ==> x != a[j];
    {
      if (a[i]==x) { return i; }
      i := i + 1;
    }
    return -1;
  }
```

Set ADT

```
class ASet {

  var a:array<int>;
  var size:int;
…
  method contains(v:int) returns (f:bool)
  requires RepInv();
  ensures  f <==> exists j::(0<=j<size) && v == a[j];
  ensures RepInv();
  {
    var p:int := find(v);
    f := (p >= 0);
  }
}
```

Construction and Verification of Software

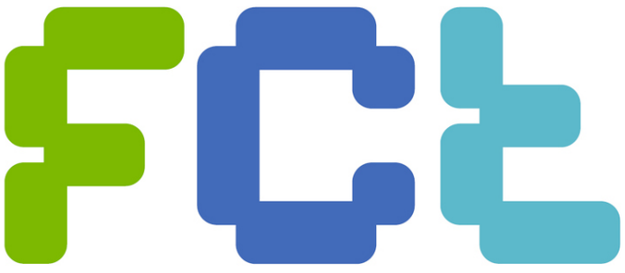
2019 - 2020

MIEI - Integrated Master in Computer Science and Informatics

Consolidation block

Lecture 4 - Part V - Soundness and Abstraction Map

João Costa Seco (joao.seco@fct.unl.pt)
based on previous editions by **Luís Caires** (lcaires@fct.unl.pt)



FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Soundness and Abstraction Map

- We have learned how to express the representation invariant and make sure that no unsound states are ever reached
- We have informally argued that the representation state in every case represents the right abstract state, but how to make sure?
- We next see how the correspondence between the representation state and the abstract state can be explicitly expressed in Dafny using ghost variables, specification operations, and abstraction map soundness check.

Technical ingredients in ADT design

RECAP

- The ***abstract state***
  - defines how client code sees the object
- The ***representation type***
  - chosen by the programmer to implement the ADT internals. The programmer is free to chose the implementation strategy (data-structures, algorithms). This is done at construction time.
- The ***concrete state***
  - in general, not all representation states are legal concrete states
  - a concrete state is a representation state that really represents some well-defined abstract state

Technical ingredients in ADT design

RECAP

- The ***representation invariant***
  - the representation invariant is a condition that restricts the representation type to the set of (safe) concrete states
  - if the ADT representation falls outside the rep invariant, something is wrong (inconsistent representation state).
- The ***abstraction function***
  - maps every concrete state into some abstract state
- The ***operation pre- post- conditions***
  - expressed for the representation type
  - also expressed for the abstract type (for client code)

Soundness and Abstraction Map

- A so-called ghost variable is only used in the spec and does not actually use memory space
- Usages of ghost variables only occur in spec operations (are never executed at runtime)

```
class ASet {
    // Abstract state
    ghost var s:set<int>;

    // Representation state
    var a:array<int>;
    var size:int;
```

- We therefore represent the abstract state with a ***ghost variable***.

Sound

- A so-ca
not act
- Usages
(are ne

```
class ASe
    // Ab
    ghost

    // Re
    var a
    var s
```

- We the

d does

ns

t variable.

Dafny - Sets

Microsoft Research

other tutorials close

Sets

1. tutorials

Sets of various types form one of the core tools of verification for Dafny. Sets represent an orderless collection of elements, without repetition. Like sequences, sets are immutable value types. This allows them to be used easily in annotations, without involving the heap, as a set cannot be modified once it has been created. A set has the type:

```
set<int>
```

for a set of integers, for example. In general, sets can be of almost any type, including objects. Concrete sets can be specified by using display notation:

load in editor

```
var s1 := {}; // the empty set
var s2 := {1, 2, 3}; // set contains exactly 1, 2, and 3
assert s2 == {1,1,2,3,3,3,3}; // same as before
var s3, s4 := {1,2}, {1,4};
```

The set formed by the display is the expected set, containing just the elements

Soundness and Abstraction Map

- We next define a boolean function `Sound()` that specifies the precise relationship the abstract and concrete state:

```
// The mapping function between abstract and representation state
function Sound():bool
    reads this,a
    requires RepInv();
{
    forall x::(x in s) <==> exists p::(0<=p<size) && (a[p] == x)
}
```

- We then express in all operations how the abstract state changes, and how it is kept well related with a proper representation state

- As a benefit, we may then also express pre and post conditions in terms of the abstract state !

Set ADT (with abstract state)

```
class ASet {
   // Abstract state
   ghost var s:set<int>;
   // Representation state
   var a:array<int>;
   var size:int;

   // The mapping function between abstract and representation state
   function Sound():bool
       reads this,a
       requires RepInv();
   { forall x::(x in s) <==> exists p::(0<=p<size) && (a[p] == x) }

   function RepInv():bool
       reads this,a
   { 0 < a.Length && 0 <= size <= a.Length && unique(a,0,size) }

   function AbsInv():bool
       reads this,a
   { RepInv() && Sound() }

   // Spec functions
   function unique(b:array<int>, l:int, h:int):bool
       reads b;
       requires 0 <= l <= h <= b.Length ;
   { forall k::(l<=k<h) ==> forall j::(k<j<h)  ==> b[k] != b[j] }
```

Set ADT (with abstract state)

```
 class ASet {
    // Abstract state
    ghost var s:set<int>;

    // Representation state
    var a:array<int>;
    var size:int;
…
    // Implementation: Constructor and Methods

    constructor(SIZE:int)
        requires SIZE > 0;
        ensures AbsInv() && s == {};
    {
        // Init of Representation state
        a := new int[SIZE];
        size := 0;
        // Init of Abstract state
        s := {};
    }
…
```

Set ADT (with abstract state)

```
class ASet {
    // Abstract state
    ghost var s:set<int>;

    // Representation state
    var a:array<int>;
    var size:int;
…
    method find(x:int) returns (r:int)
        requires AbsInv()
        ensures  AbsInv()
        ensures -1 <= r < size;
        ensures r < 0 ==> forall j::(0<=j<size) ==> x != a[j];
        ensures r >=0 ==> a[r] == x;
    {
        var i:int := 0;
        while (i<size)
            decreases size-i
            invariant 0 <= i <= size;
            invariant forall j::(0<=j<i) ==> x != a[j];
        {
            if (a[i]==x) { return i; }
            i := i + 1;
        }
        return -1;
    }
…
```

Set ADT (with abstract state)

```
class ASet {
    // Abstract state
    ghost var s:set<int>;

    // Representation state
    var a:array<int>;
    var size:int;
…
    method add(x:int)
        modifies a, this
        requires AbsInv()
        requires count() < maxsize()
        ensures  AbsInv() && s == old(s) + {x}
    {
        var i := find(x);
        if (i < 0) {
            a[size] := x;
            s := s + { x };
            size := size + 1;
            assert a[size-1] == x;
            assert forall i :: (0<=i<size-1) ==> (a[i] == old(a[i]));
            assert forall x::(x in s) <==> exists p::(0<=p<size) && (a[p] == x);
        }
    }
…
```